Technische Universität Berlin

TKN

Praktikum

Rechnernetze

Projekt 2:

Distributed Hash Table

In diesem Projekt implementieren Sie eine Distributed Hash Table (DHT) basierend auf dem Chord-Protokoll. Ihre finale Abgabe wird anhand von automatisierten Tests bewertet.

Hinweise

Die praktischen Aufgaben sind in Kleingruppen von bis zu vier Personen zu lösen. Reichen Sie deshalb Ihren Quelltext bzw. Lösungen dieser Aufgaben bis zum **06.01.2023 23:59 Uhr per ISIS** ein. Aufgaben werden automatisch sowohl auf Plagiate, als auch auf Korrektheit getestet, halten Sie sich deshalb genau an das vorgegebene Abgabeformat.

Auf der ISIS-Seite zur Veranstaltung finden Sie zusätzliche Literatur und Hilfen, insbesondere den *Beej's Guide to Network Programming Using Internet Sockets*[1], für die Bearbeitung der Aufgaben!

Einleitung: Hash Table

In einer Hash Table werden Tupel der Form `<key, value>` gespeichert [2]. Dabei bestimmt der Hash-Wert, der auf Basis des `key`s berechnet wird, kurz `hash(key)`, in welcher Zeile der Tabelle der `value` gespeichert wird. Eine Hash-Tabelle ist im einfachsten Fall auf einem einzelnen Server gespeichert und wird von mehreren Clients verwendet. Dazu gibt es die Befehle `set()`, um einen Wert zu speichern oder zu aktualisieren, `get()`, um einen Wert auszulesen, und `delete()`, um einen Wert zu löschen. Das Nutzen eines einzelnen Servers zur Speicherung hat diverse Nachteile, u.a. kann der Server überlastet sein bei zu vielen parallelen Anfragen.

[1] https://beej.us/guide/bgnet

[2] https://de.wikipedia.org/wiki/Hashtabelle

In der Vorlesung wurde Ihnen als Alternative das Chord-Protokoll vorgestellt. Damit lässt sich eine Hash-Tabelle dezentralisiert, d.h. über mehrere Server (sog. Knoten oder Peers) verteilt, speichern und als Distributed Hash Table (kurz: DHT) betreiben. Im Unterschied zur Hash-Tabelle auf einem einzelnen Server wird hier der Wert `hash(key)` verwendet, um den Peer zu ermittlen, auf dem der `value` gespeichert wird. Ziel dieses Projekts ist es, einen Server zu entwickeln, der als Peer in einem Chord-Ring fungieren kann.

Einarbeitung Code-Skelett

Nutzen Sie die Beschreibung im Anhang A, um sich einen abstrakten Überblick über die Funktionalität des mitgelieferten Code-Skeletts zu verschaffen. Kompilieren Sie anschließend den Code:

```
$ cmake -B build -DCMAKE_BUILD_TYPE=Debug
$ make -C build
```

Installieren Sie nun die Testbench (benötigt `docker` und ggf. `docker pull ubuntu:bionic`) für dieses Projekt aus dem entpackten Ordner und lassen Sie anschließend die Tests laufen:

```
$ pip install --force-reinstall rnvs_tb-2022_projekt2_1-py3-none-any.whl
$ sudo rnvs-tb-dht -s .
```

Sie werden feststellen, dass Tests fehlschlagen. Nutzen Sie diese fehlgeschlagenen Tests für ein testgetriebenes Entwickeln [3]. Beachten Sie, dass sowohl das Client- als auch das Peer-Programm im Rahmen dieser Abgabe auf ein und derselben Maschine laufen (Ihrem lokalen Rechner). Im Code-Skelett sehen Sie, dass die Trennung von Client und Peer mithilfe von Prozessen und Sockets abgebildet wird.

Implementierung Distributed Hash Table (DHT)

Implementieren Sie eine Distributed Hash Table (DHT), mit der `<key, value>`-Paare über mehrere Prozesse bzw. Rechner verteilt gespeichert werden. Nutzen Sie als Ausgangspunkt

[3] https://de.wikipedia.org/wiki/Testgetriebene_Entwicklung

das mitgelieferte Code-Skelett, dass Sie an den mit `/* TODO IMPLEMENT */` gekennzeichneten Stellen ergänzen sollen, sodass die mitgelieferten Tests nicht mehr fehlschlagen. Konkret müssen Sie innerhalb der folgenden Dateien bereits vorhandene Funktionen mit weiterem Code ergänzen:

* `peer.c`
* `neighbor.c`
* `hash_table.c`

Weitere Details zu der erwarteten Funktion der Methoden finden Sie in der zugehörigen Dokumentation.

Die Speicherung der Werte soll eine DHT auf Basis von einem vereinfachten Chord-Ring übernehmen. Eine DHT ist dabei ein Peer-to-Peer (P2P) Netzwerk, in dem alle Knoten der DHT als gleichwertige Peers fungieren, die sowohl Anfragen entgegennehmen als auch weiterleiten können. Die Interaktion eines anfragenden Clients sowie der Peers der DHT soll für eine beispielhafte Topologie aus vier Peers wie folgt aussehen:

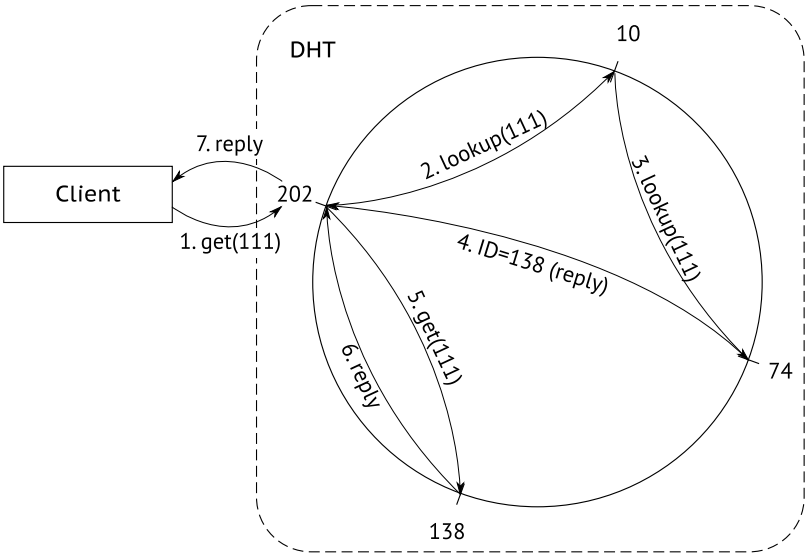


Figure 1: Beispielhafte Abfrage eines Keys in der DHT

Eine entsprechende Topologie wird mithilfe des Skriptes `launch.sh` bereitgestellt und enthält im Wesentlichen folgende Befehle:

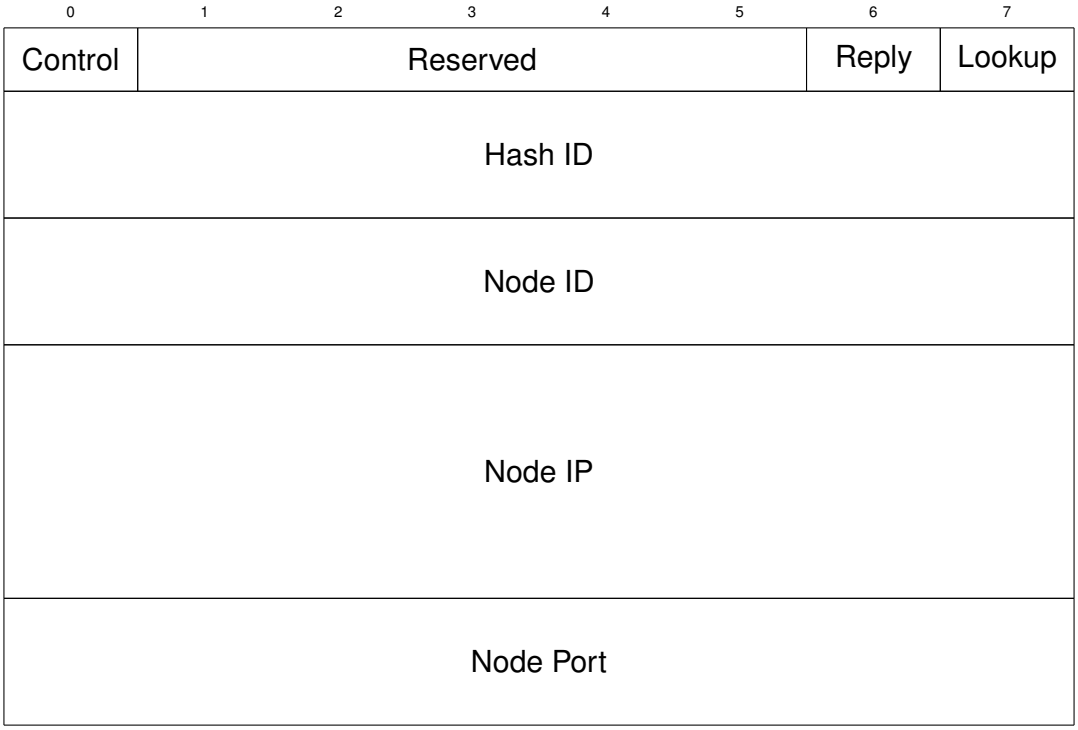
```
./peer \
    202 127.0.0.1 4711 \  # Eigene Adresse
    138 127.0.0.1 4710 \  # Vorgänger
    10 127.0.0.1 4712     # Nachfolger

./peer 138 127.0.0.1 4710 74 127.0.0.1 4713 202 127.0.0.1 4711
./peer 74 127.0.0.1 4713 10 127.0.0.1 4712 138 127.0.0.1 4710
./peer 10 127.0.0.1 4712 202 127.0.0.1 4711 74 127.0.0.1 4713
```

Jeder Peer bekommt beim Aufruf sein eigenes Tripel `<ID, IP, Port>`, dann das Tripel `<ID, IP, Port>` des Vorgängers und schließlich das Tripel `<ID, IP, PORT>` des Nachfolgers übergeben. So weiß jeder Peer, für welchen Bereich er zuständig ist und wer seine Vor- und Nachfolger sind.

Das Paketformat für *Anfragen* (`get`, `set` und `delete`) und deren Antworten wird im Anhang A beschrieben. Die Peers des Chord-Rings bearbeiten diese Anfragen unabhängig davon, ob diese von einem vollständigem Mitglied des Rings, oder einer beliebigen anderen Quelle gestellt werden.

Zusätzlich zu den Anfragen gibt es *Kontrollnachrichten* für den Lookup bzw. die entsprechende Antwort. Das Nachrichtenformat für die Kontrollnachrichten sieht wie folgt aus (Anhang B erläutert, wie die Darstellung zu interpretieren ist):



Dabei ist `Control` der Indikator für eine Kontrollnachricht innerhalb des Rings, also einen Lookup oder dessen Antwort. Die IPv4-, Port-, und ID-Felder sind beim Lookup mit den Daten des ersten Knotens zu füllen, der den Befehl von außen erhalten hat und jetzt stellvertretend nach dem zuständigen Knoten sucht. Achten sie darauf, dass Zahlen stets in Network-Byte-Order[4] versendet werden. Wenn ein Lookup beantwortet wird, schreibt der antwortende Peer jeweils die ID, die Adresse und den Port des zuständigen Knotens in die Antwort.

Abgabe

Die vorgegebene `CMakeLists.txt` ist so konfiguriert, dass Sie damit ein Archiv des Projekts erstellen können:

```
make -Cbuild package_source
```

Das generierte Archiv sollte nun im `build`-Ordner verfügbar sein: `RN-Praxis2-0.1.1-Source.tar.gz`
Laden Sie Ihre so generierte Abgabe bis zum **06.01.2023, 23:59 CET** auf ISIS hoch.

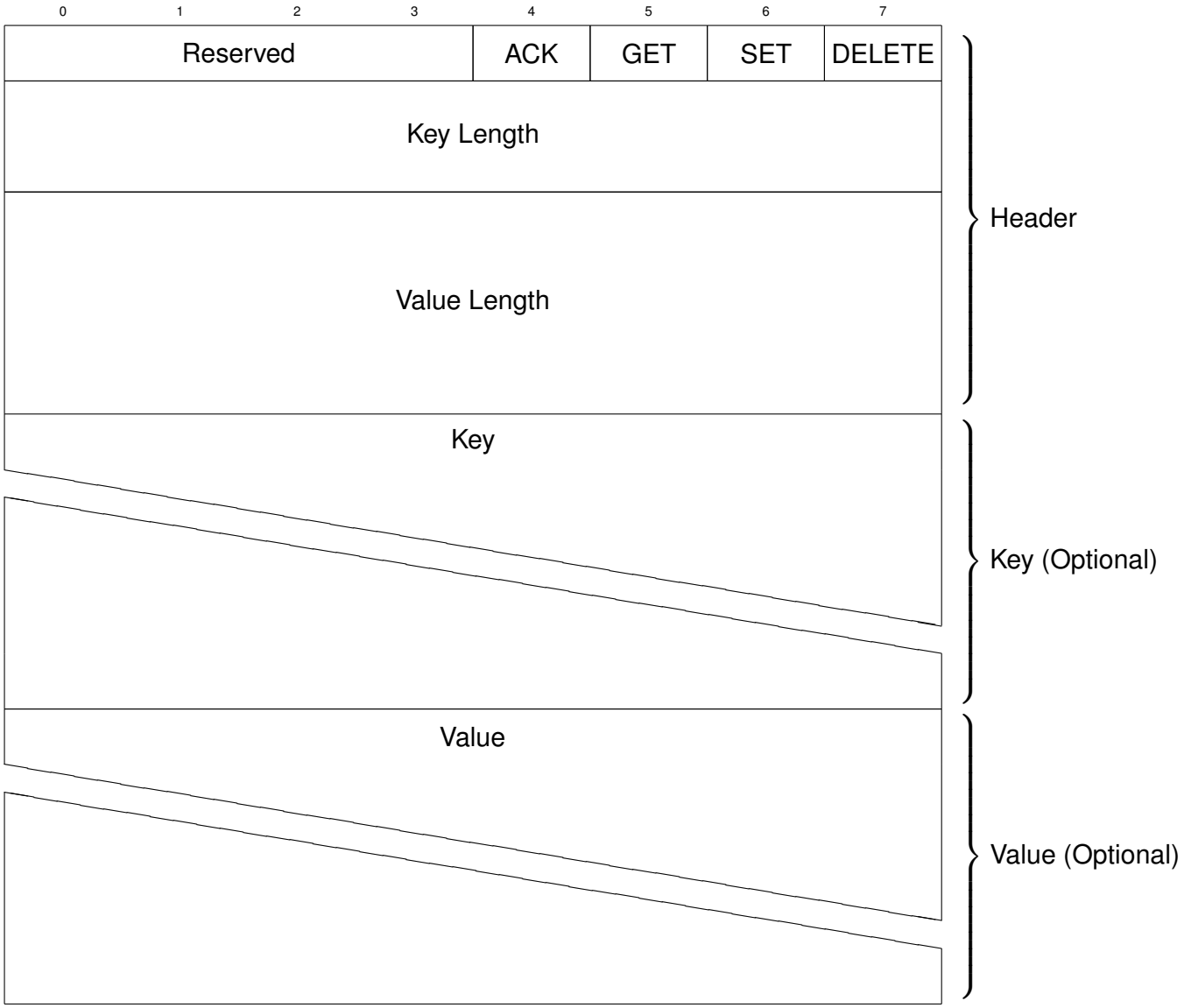
[4] https://de.wikipedia.org/wiki/Byte-Reihenfolge#Plattform%C3%BCbergreifende_Darstellung_von_Zahlen

Abgaben in andern Formaten werden nicht akzeptiert bzw. mit 0 Punkten bewertet!

Viel Erfolg :)

Anhang A: Hash Table Client/Peer-Architektur

Das Code-Skelett beschreibt eine Client/Peer-Architektur, in der Clients Werte in einer externen Hash-Table speichern, abrufen oder löschen können. Dazu kommunizieren die Clients mit einem beliebigen Peer aus einem Chord-Ring, der die Chord-spezifischen Anfragen übernimmt und sich dem Client ggü. wie ein Server verhält. Das Protokoll basiert auf TCP, d.h. die unten angegebenen Nachrichten werden über einen zuverlässigen Bytestrom geschickt. Das Nachrichtenformat ist aufgeteilt in einen festen Header, den Key mit variabler Länge und einen optionalen Value variabler Länge.



Der Header enthält zunächst einige reservierte Bits. Diese sind standardmäßig mit Nullen gefüllt werden und könnten später z.B. für ein Versions-Feld benutzt werden. Danach folgen Bits für die Befehle bzw. die Server-Antwort. Der Client kann hier das GET, SET oder DELETE Bit setzen. Ein gesetztes Bit bedeutet, dass es sich bei der Nachricht um diesen Anfragetyp handelt. Der Client ist so programmiert, dass immer nur ein Bit gesetzt ist. Der Server bestätigt alle Befehle bei erfolgreicher Durchführung mit einem Acknowledgment (`ACK`) und setzt dafür das sowohl das entsprechende Bit, als auch das Bit der erfolgreich durchgeführten Aktion. Diese Bestätigung erlaubt dem Client die Zuordnung des Acknowledgments zu der entsprechenden Anfrage.

Nun folgt die Länge des Keys als 16 bit vorzeichenlose Ganzzahl und die Länge des Values als 32 bit vorzeichenlose Ganzzahl. Alle Angaben beziehen sich, falls nicht anders angegeben, auf die Network-Byte-Order! `GET`- und `DELETE`-Anfragen enthalten nur einen Key, so dass in beiden Fällen die Länge des Values auf `0` gesetzt wird. Antworten auf `GET`-Anfragen enthalten Key und Value, die anderen Antworten enthalten weder Key noch Value. Nach dem Header wird je nach Anfrage bzw. Antwort der Key gesendet, der variabler Länge ist. Darauf folgt optional der Value, der ebenfalls variabler Länge ist.

Client

Der Client bekommt als Kommandozeilenargumente den DNS-Namen bzw. die IP-Adresse des Servers, die Methode (`SET`, `GET`, `DELETE`) und den Key übergeben. Im Falle eines `GET`s wird der Value zurückgegeben, sonst nichts. Je nach Methode soll der Client von der Kommandozeile lesen und die Daten entsprechend als Value verwenden. Damit lässt sich dann bspw. eine Art Pseudo-Dateisystem nachbilden, indem Dateien mittels einer Pipe an ihren Client übergeben werden.

Einige Beispielaufrufe dazu:

```
# Store `cat.jpg` in DHT
./client localhost 4711 SET /pics/cat.jpg < cat.jpg

# Query stored image
./client localhost 4711 GET /pics/cat.jpg > cat2.jpg
```

```
# Remove image from DHT
./client localhost 4711 DELETE /pics/cat.jpg
```

Hinweise:

- Der Value muss nicht notwendigerweise eine Datei sein, sondern kann beliebige Bytes enthalten.
- < und > sind keine Parameter, sondern Shell redirections[5]! Der Aufruf

  ```
  cat 'cat.jpg' | ./client localhost 4711 SET /pics/cat.jpg
  ```

  wäre also äquivalent zum ersten Beispiel.
- Die Keys (im Beispiel: `/pics/cat.jpg`) beziehen sich *nicht* auf den lokalen Rechner, sondern die Daten im Chord-Ring.

Hashing Bibliothek

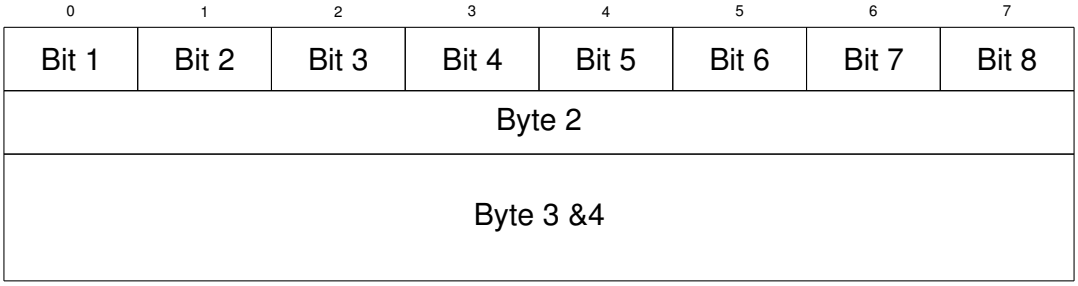
Im Code-Skelett wird als externe Abhängigkeit die Bibliothek `uthash` [6] verwendet. Machen Sie sich hier insbesondere mit den Makros `HASH_FIND` und `HASH_ADD_KEYPTR` vertraut, die im Skelett verwendet werden. Nutzen Sie hier die Suchfunktion Ihres Browsers, um schnell zu den für Sie relevanten Stellen zu gelangen. Es wird im Laufe Ihrer Karriere sicher häufig vorkommen, dass Sie sich schnell in der Dokumentation externer Bibliotheken zurechtfinden müssen.

Anhang B: Darstellung von Paketen

Netzwerkprotokolle enthalten häufig Darstellungen, die die Paketformate beschreiben. Diese sind zwar nicht standardisiert, folgen aber immer dem gleichen Schema: Die Bytefolge in Zeilen aus `n`-Bytes (häufig 1, 2 oder 4) gebrochen und als Tabelle dargestellt. Bits, welche zusammen gehören werden als ein Kasten dargestellt und mit dem entsprechenden

[5] https://www.gnu.org/software/bash/manual/html_node/Redirections.html
[6] https://troydhanson.github.io/uthash/userguide.html

Feldnamen des Protokolls versehen. Als Beispiel geben wir im Folgenden ein Paket sowohl in der oben beschriebenen Darstellung, als auch als Datenstrukturen die in C verwendet würden um dieses zu repräsentatieren an.




```c
enum Bits {
    BIT_1 = 1 << 0,
    BIT_2 = 1 << 1,
    BIT_3 = 1 << 2,
    BIT_4 = 1 << 3,
    BIT_5 = 1 << 4,
    BIT_6 = 1 << 5,
    BIT_7 = 1 << 6,
    BIT_8 = 1 << 7,
};

struct Packet {
    enum Bits byte1;
    uint8_t byte2;
    uint16_t byte3_and_4;
};
```